EECS 22: Advanced C Programming

Lecture 3

Rainer Dömer

doemer@uci.edu

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

Lecture 3: Overview

- Review of the C Programming Language
  - Operators and Expressions
    - Arithmetic, Increment, Decrement, Assignment
    - Relational, Logical, Bitwise, Shift, Conditional
    - Others
  - Operator Precedence and Associativity

Operators in C

- Arithmetic Operators
- Increment and Decrement Operators
- Assignment Operator
- Augmented Assignment Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Shift Operators
- Conditional Operator
- Other Operators

Arithmetic Operators

- Arithmetic Operators
  - parentheses **(**, **)**
  - unary plus, minus **+**, **-**
  - multiplication, division, modulo *****, **/**, **%**
  - addition, subtraction **+**, **-**
- Evaluation order of expressions
  - binary operators evaluate left to right
  - unary operators evaluate right to left
  - by operator precedence
    - ordered as in table above (higher operators are evaluated first)
- Arithmetic operators are available
  - for integer types: all
  - for floating point types: all except **%**

Increment and Decrement Operators

- Counting in steps of one
  - increment (add 1)
  - decrement (subtract 1)
- C provides special counting operators
  - increment operator: **++**
    - **count++** post-increment (**count = count + 1**)
    - **++count** pre-increment (**count = count + 1**)
  - decrement operator: **--**
    - **count--** post-decrement (**count = count - 1**)
    - **--count** pre-decrement (**count = count - 1**)
  - Note: Argument must be an integral *lvalue*!
    - **Lvalue**: an expression referring to an object (i.e. variable name)
    - An *lvalue* can be used as the *left* argument for an assignment!

Increment and Decrement Operators

- Difference between Pre- and Post- Operators
  - *pre-* increment/decrement
    - value returned is the incremented/decremented (new) value
  - *post-* increment/decrement
    - value returned is the original (old) value
  - Examples:

```
int n = 5;
int x = 0;
x = n++;
```

➢ x = 5
➢ n = 6

```
int n = 5;
int x = 0;
x = ++n;
```

➢ x = 6
➢ n = 6

Assignment Operator

- Assignment operator: **=**
  - evaluates right-hand argument
  - assigns result to left-hand argument
    - Evaluation order: right-to-left!
  - Left-hand argument must be a lvalue
  - Result is the new value of left-hand argument
- Example:
  - `int a, b, c;`
  - `int d = 5;  /* initialization, not an assignment */`
  - `a = 42;  /* assignment */`
  - `b = c = 0;  /* same as c = 0; b = c; */`

Augmented Assignment Operators

- Augmented assignment operators: **+=**, ***=**, ...
  - evaluates right-hand side as temporary result
  - applies operation to left-hand side and temporary result
  - assigns result of operation to left-hand side
    - Evaluation order: right-to-left!
  - Left-hand argument must be a lvalue
- Example: Counter
  - `int c = 0;  /* counter starting from 0 */`
  - `c = c + 1;  /* counting by regular assignment */`
  - `c += 1;  /* counting by augmented assignment */`
- Augmented assignment operators:
  - **+=**, **-=**, ***=**, **/=**, **%=**, **<<=**, **>>=**, **|=**, **^=**, **&=**

Relational Operators

- Comparison of values
  - **<** less than
  - **>** greater than
  - **<=** less than or equal to
  - **>=** greater than or equal to
  - **==** equal to (remember, **=** means assignment!)
  - **!=** not equal to
- Relational operators are defined for all basic types
  - integer (e.g. **`5 < 6`**)
  - floating point (e.g. **`7.0 < 7e1`**)
- Result type is Boolean, but represented as integer
  - false **`0`**
  - true **`1`** (or any other value *not* equal to zero)

Logical Operators

- Operation on Boolean (truth) values
  - **!** "not" logical negation
  - **&&** "and" logical and
  - **||** "or" logical or
- Truth table:

x	y	!x	x && y	x || y
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

- Argument and result types are Boolean, but represented as integer
  - false **`0`**
  - true **`1`** (or any other value *not* equal to zero)

Logical Operators

- *Lazy* evaluation for logical *and* and logical *or*
  - Evaluation order left-to-right
  - Logical *and* has higher priority than logical *or*
  - Expression evaluation stops as soon as the result is known
    - Logical *and* evaluates right-hand argument only if left-hand is true (1)
    - Logical *or* evaluates right-hand argument only if left-hand is false (0)
  - Example:
    - `v = f() && g() || h();`
    - Function `f()` is called first
    - Function `g()` is called only if `f()` returned `1`
    - Function `h()` is called only if result of `f()&&g()` returned `0`
  - Exercise:
    - Is it possible that only `f()` and `h()` are called?

Bitwise Operators

- Operators for bit manipulation
  - `&` bitwise "and" `0xFF & 0xF0 = 0xF0`
  - `|` bitwise inclusive "or" `0xFF | 0xF0 = 0xFF`
  - `^` bitwise exclusive "or" `0xFF ^ 0xF0 = 0x0F`
  - `~` bitwise negation (one's complement) `~0xF0 = 0x0F`
  - `<<` left shift `0x0F << 4 = 0xF0`
  - `>>` right shift `0xF0 >> 4 = 0x0F`
  - ➢ Bitwise operators are only available for integral types
- Typical usage
  - Mask out some bits from a value
    - `c = c & 0x0F` extracts lowest 4 bits from `char c`
  - Set a set of bits in a value
    - `c = c | 0x0F` sets lowest 4 bits of `char c`

Shift Operators

- Left-shift operator: **`x << n`**
  - shifts *x* in binary representation *n* times to the left
  - ➢ multiplies *x* *n* times by 2
  - Examples
    - $2x$ = **`x << 1`**
    - $4x$ = **`x << 2`**
    - $x * 2^n$ = **`x << n`**
    - 2^n = **`1 << n`**
- Right-shift operator: **`x >> n`**
  - shifts *x* in binary representation *n* times to the right
  - ➢ divides *x* *n* times by 2
  - Examples
    - $x / 2$ = **`x >> 1`**
    - $x / 4$ = **`x >> 2`**
    - $x / 2^n$ = **`x >> n`**

Conditional Operator

- Conditional evaluation of values in expressions
- Question-mark operator:
  `test ? true-value : false-value`
  - evaluates the ***`test`***
  - if ***`test`*** is true, then the result is ***`true-value`***
  - otherwise, the result is ***`false-value`***
- Examples:
  - **`(4 < 5) ? (42) : (4+8)`** evaluates to **`42`**
  - **`(2==1+2) ? (x) : (y)`** evaluates to **`y`**
  - **`(x < 0) ? (-x) : (x)`** evaluates to **`abs(x)`**
- Note: Exactly one of the two cases is evaluated
  - Example: **`Test() ? f() : g();`**
    If **`Test()`** returns true, **`f()`** is called, otherwise **`g()`**

Other Operators

- Comma operator: ***expr1, expr2***
  - Left-to-right evaluation, result is result of right operand
- Array access operator: ***expr1*[*expr2*]**
  - Detailed discussion in Lecture 5
- Function call: ***expr1*(*expr2*)**
  - Detailed discussion in Lectures 6 and 7
- Member access: ***expr1*.*expr2***, ***expr1*->*expr2***
  - Detailed discussion in Lectures 16 and 17
- Pointer operators: **&*expr***, ****expr***
  - Detailed discussion in Lectures 17 and later
- Type casting: **(*typename*) *expr***
  - Detailed discussion in Lecture 26

Operator Precedence and Associativity

– parenthesis, array/member acc.	**()**, **[]**, **.**, **->**	left to right
– unary operators, pointer op., size of, type cast	**!**, **~**, **++**, **--**, **+**, **-**, *****, **&**, **sizeof**, **(*typename*)**	right to left
– multiplication, division, modulo	*****, **/**, **%**	left to right
– addition, subtraction	**+**, **-**	left to right
– shift left, shift right	**<<**, **>>**	left to right
– relational operators	**<**, **<=**, **>=**, **>**	left to right
– equality	**==**, **!=**	left to right
– bitwise and	**&**	left to right
– bitwise exclusive or	**^**	left to right
– bitwise inclusive or	**|**	left to right
– logical and	**&&**	left to right
– logical or	**||**	left to right
– conditional operator	**?:**	left to right
– assignment operators	**=**, **+=**, **-=**, ***=**, **/=**, …	right to left
– comma operator	,	left to right